

Runtime Verification with Ogma

Ivan Perez, Ph.D

`ivan.perezdominguez@nasa.gov`

KBR @ NASA Ames Research Center

January 19, 2023

Invited talk

University of California Santa Cruz

Runtime Verification

- ▶ Technique for monitoring systems as they run, and detect property violations.
 - ▶ Unexpected behavior of the system under study.
 - ▶ Unexpected behavior of the environment.
- ▶ Online or offline.
- ▶ Normally based on temporal logic.

See: Havelund and Goldberg, "Verify Your Runs". 2008.

Domain of Interest

- ▶ Safety-critical systems
- ▶ Aircraft, spacecraft
- ▶ Embedded systems

Goals

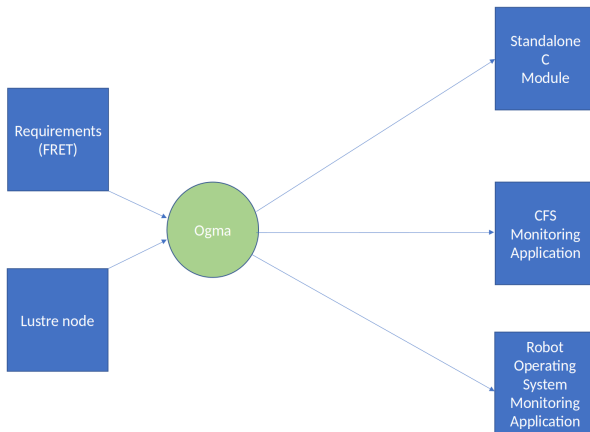
- ▶ High-level
- ▶ Verifiably correct code
- ▶ Real time
- ▶ Ease of integration

The importance of using a high-level language

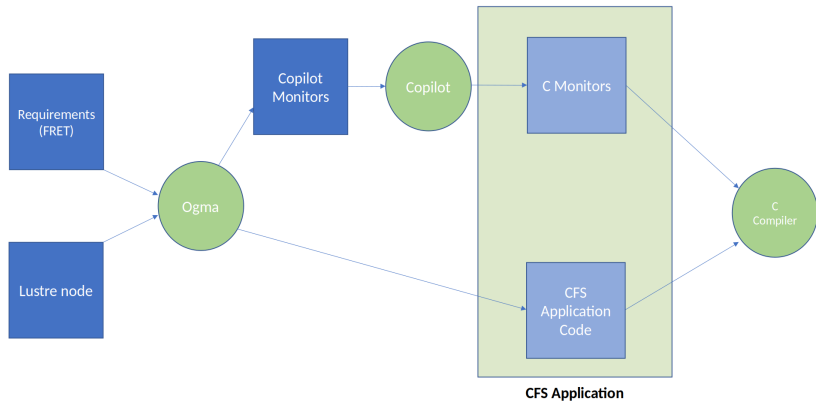
- ▶ Errors in the RV system can threaten the mission as a whole.
- ▶ Low-level languages may be more error-prone, and some classes of errors are easier to make.
- ▶ A high-level, safe language can facilitate readability and maintenance, and limit the likelihood of introducing (some) bugs.

See: Ray et al., "A Large Scale Study of Programming Languages and Code Quality in Github", 2014.

Approach

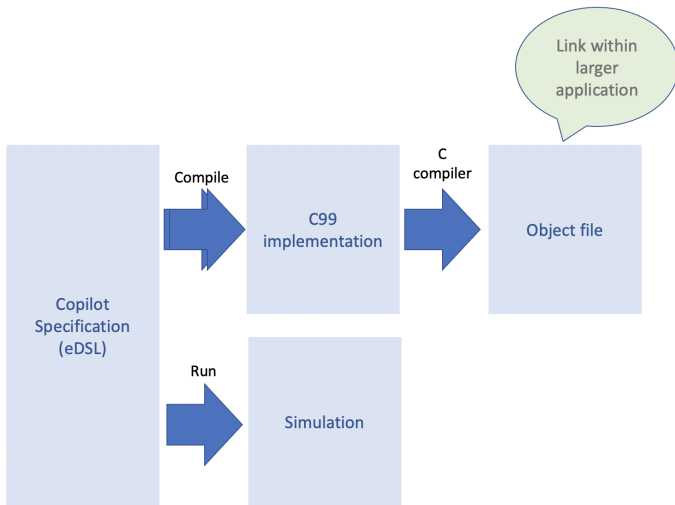


Under the hood



- ▶ High-level Runtime Verification framework that produces hard real-time C99.

Copilot workflow



Structure of a Copilot module

- ▶ External data

```
sensorData = extern "global_c_value" Nothing
```

- ▶ Properties to monitor















```
undesirableProperty = sensorData >= 10
```

- ▶ Triggers

```
spec = do  
  trigger "global_c_handler" undesirableProperty []
```

Property language

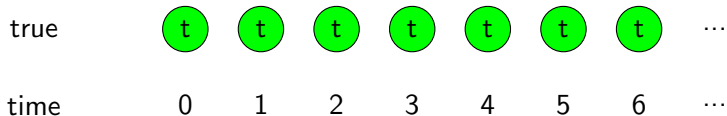
- ▶ Primitives and combinators
- ▶ Based on streams

true								...
signal								...
time	0	1	2	3	4	5	6	...






















Property Language: Structure

(stream name) — ^{name = expression} — (stream definition)

Property Language: Primitives: true



Property Language: Point-wise combinators: Boolean logic

p1								...
p2								...
p1 && p2								...
time	0	1	2	3	4	5	6	...

Property Language: Primitives: addition

p1	0	1	0	2	0	3	0	...
p2	5	1	5	1	5	1	5	...
p1 + p2	5	2	5	3	5	4	5	...
time	0	1	2	3	4	5	6	...

Property Language: Primitives: number overloading

p1	0	1	0	2	0	3	0	...
1	1	1	1	1	1	1	1	...
p1 + 1	1	2	1	3	1	4	1	...
time	0	1	2	3	4	5	6	...

The expression 1 is overloaded to mean both the number 1 in one sample, and the stream that has 1 at every sample.

Property Language: Point-wise operators

- ▶ Logic: $\&\&$, $\|$, *not*, \implies , ...
- ▶ Comparison: $<$, $>$, \leq , \geq , $==$, $/=$, ...
- ▶ Arithmetic: $+$, $-$, $*$, $/$, ...
- ▶ Trigonometry: *sin*, *cos*, *tan*, ...

Property Language: Temporal translations: delays

fiveThenCount = [5] ++ count

count	0	1	2	3	4	5	6	...
-------	---	---	---	---	---	---	---	-----

fiveThenCount	5	0	1	2	3	4	5	...
---------------	---	---	---	---	---	---	---	-----

time	0	1	2	3	4	5	6	...
------	---	---	---	---	---	---	---	-----

Property Language: Temporal translations: delays

`insert2ThenCount = [5, 10] ++ count`

`drop1ThenCount = drop 1 insert2ThenCount`

<code>insert2ThenCount</code>	5	10	0	1	2	3	4	...
<code>drop1ThenCount</code>	10	0	1	2	3	4	5	...
time	0	1	2	3	4	5	6	...

Property Language: Recursion

counter	0	1	2	3	4	5	6	...
time	0	1	2	3	4	5	6	...

`counter = [0] ++ (counter + 1)`

`counter = [0] ++ (([0] ++ (counter + 1)) + 1)` — expand counter

`counter = [0] ++ ([0+1] ++ (counter + 1 + 1))` — distribute (+1)

`counter = [0] ++ ([1] ++ (counter + 2))` — apply additions

`counter = [0, 1] ++ (counter + 2)` — associat. append

Property Language: Temporal Logics

```
prop  = PTLTL.alwaysBeen (counter <= 4)
prop2 = (MTL.alwaysBeen 0 3 (temperature >= 100))
      && (MTL.alwaysBeen 0 40 (airspeed >= 100))

recover = (MTL.eventuallyPrev 0 100 (airspeed < 100))
      && (MTL.alwaysBeen 0 10 (airspeed >= 100))
```

- ▶ Past-Time Linear Temporal Logic
- ▶ Metric Temporal Logic
- ▶ Bounded Linear Temporal Logic

Property Language: Other libraries

- ▶ Voting (used for fault tolerance)
- ▶ Statistics
- ▶ Clocks (ticking at different rates)
- ▶ Stack machines
- ▶ Regexp recognition

Property Language: Externs

Copilot

```
extVar = extern "global_var" Nothing
```

C

```
int global_var = 0;
```

```
int main (...) {  
    // Sense data  
    global_var = sensing_operation();  
    // Check monitors  
    step();  
}
```

Property Language: Copilot monitors can be much more complex

```
counter :: Stream Int32
counter = [0] ++ (counter + 1)
```

```
elevation :: Stream Double
elevation = extern "elevation" Nothing
```

— Estimate derivative by delaying the elevation

```
climbrate :: Stream Double
climbrate = elevation - ([0] ++ elevation)
```

— Specification that defines triggers based on streams

```
spec :: Spec
spec = do
  let falling :: Stream Bool
      falling = climbrate < (-6)
```

```
trigger "falling"      falling [arg counter, arg climbrate]
trigger "not_fallen" (alwaysBeen $ not falling) []
```

Arrays

```
temps :: Stream (Array 3 Float)
temps = constant (array [23.2,24.0,23.5])
```

```
temp2 :: Stream Float
temp2 = temps .!! 2    — 23.5
```

Arrays

```
a1 :: Array 3 Int8
```

```
a1 = array [1,2,3]
```

```
a2 :: Array 2 Int8
```

```
a2 = a1
```

```
Main.hs:16:6: error:
```

```
  * Couldn't match type '3' with '2'
```

```
    Expected type: Array 2 Int8
```

```
    Actual type: Array 3 Int8
```

```
  * In the expression: a1
```

```
    In an equation for 'a2': a2 = a1
```

Structs

```
data Vec = Vec { x :: Field "x" Float  
                , y :: Field "y" Float }
```

```
sensorVec :: Stream Vec  
sensorVec = extern "vector" Nothing
```

```
sensorX :: Stream Float  
sensorX = sensorVec # x
```

A minimal example: Copilot

```
import Copilot.Language
import Copilot.Language.C99
import qualified Prelude hiding (<)

sampleSensor :: Stream Float
sampleSensor = extern "sample_sensor" Nothing

property :: Stream Bool
property = sampleSensor < 10

spec :: Spec
spec = do
  trigger "handler" property []

main = do
  r <- reify spec
  compile "example" r
```

A minimal example: C

```
#include "example.h"
```

```
float sample_sensor = 0.0;
```

```
void handler(void) {  
    ... // handle property here  
}
```

```
int main() {  
    while (1) {  
        sample_sensor = ...; some operation to refresh sensor value  
        step();  
    }  
}
```

Installation

Debian ≥ 12 / Ubuntu ≥ 23.04 :

```
$ sudo apt-get install libghc-copilot-dev
```

Debian < 12 / Ubuntu < 23.04 :

```
$ sudo apt-get install cabal-install ghc
```

```
$ cabal update
```

```
$ cabal install --lib copilot
```

Mac:

```
$ brew install cabal-install
```

```
$ cabal update
```

```
$ cabal install --lib copilot
```

Compiling Copilot into C99

```
$ runhaskell Monitor.hs  
$ ls  
example.c example.h example_types.h
```

From requirements to autonomous flight

- ▶ Requirements elicitation (natural language).
- ▶ Transform requirements into Temporal Logic formulas.
- ▶ Transform Temporal Logic formulas into runtime monitors.
- ▶ Generate hard real-time code for monitors.

Distance between requirements and Temporal Logic

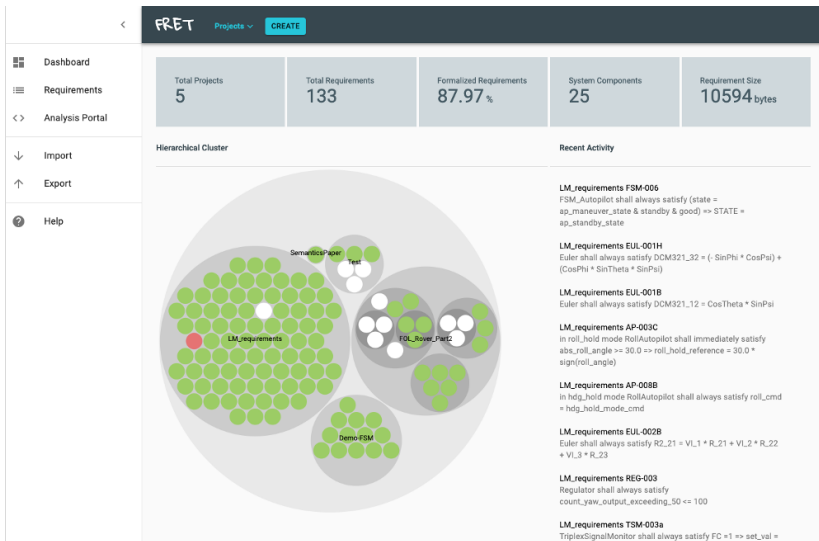
“While flying, if the airspeed drops below 100m/s, the autopilot shall increase the airspeed above 100m/s in less than 10 seconds.”

What is the temporal logic formula?

From requirements to autonomous flight (II)

Goal: close the gap between requirements and Copilot monitors.

Tool for requirements elicitation developed at NASA Ames.



FRETish

- ▶ FRET requirements are expressed in structured natural language (FRETish):

`scope condition component* shall* timing response*`

- ▶ Example:

NL: *“While flying, if the airspeed is below 100 m/s, the autopilot shall increase the airspeed to at least 100 m/s within 10 seconds.”*

FRETish: `in flight mode if airspeed < 100 the aircraft shall within 10 seconds satisfy (airspeed >= 100)`

From FRETish to TL

- FRET produces a past-time temporal logic formula for the requirement:

NL: “While flying, if the airspeed is below 100 m/s, the autopilot shall increase the airspeed to at least 100 m/s within 10 seconds.”

FRETish: in flight mode if airspeed < 100 the aircraft shall within 10 seconds satisfy (airspeed ≥ 100)

pmLTL: $H (\text{Lin_flight} \rightarrow (Y (((O_{[=10]} (((\text{airspeed} < 100) \ \& \ ((Y \neg (\text{airspeed} < 100)))) \mid \text{Fin_flight})) \ \& \ \neg (\text{airspeed} \geq 100)))) \rightarrow (O_{[<10]} (\text{Fin_flight} \mid (\text{airspeed} \geq 100)))) S (((O_{[=10]} (((\text{airspeed} < 100) \ \& \ ((Y \neg (\text{airspeed} < 100)))) \mid \text{Fin_flight})) \ \& \ \neg (\text{airspeed} \geq 100)))) \rightarrow (O_{[<10]} (\text{Fin_flight} \mid (\text{airspeed} \geq 100)))) \ \& \ \text{Fin_flight})))) \ \& \ ((\neg \text{Lin_flight}) S ((\neg \text{Lin_flight}) \ \& \ \text{Fin_flight})) \rightarrow (((O_{[=10]} (((\text{airspeed} < 100) \ \& \ ((Y \neg (\text{airspeed} < 100)))) \mid \text{Fin_flight})) \ \& \ \neg (\text{airspeed} \geq 100)))) \rightarrow (O_{[<10]} (\text{Fin_flight} \mid (\text{airspeed} \geq 100)))) S (((O_{[=10]} (((\text{airspeed} < 100) \ \& \ ((Y \neg (\text{airspeed} < 100)))) \mid \text{Fin_flight})) \ \& \ \neg (\text{airspeed} \geq 100)))) \rightarrow (O_{[<10]} (\text{Fin_flight} \mid (\text{airspeed} \geq 100)))) \ \& \ \text{Fin_flight}))),$

where *Fin_flight* (First timepoint in flight mode) is *flight* & (FTP | Y !flight), *Lin_flight* (Last timepoint in flight mode) is !flight & Y flight, FTP (First Time Point) is ! Y true.

Ogma: From FRET's TL to Copilot

- ▶ Ogma is an open-source NASA tool that transforms high level specifications (FRET component specifications, Lustre node specifications) into monitoring applications.
- ▶ Ogma has 3 modes of operation: producing standalone Copilot monitors, producing NASA Core Flight System applications, and producing Robot Operating System (ROS2) monitoring packages.
- ▶ The standalone Copilot module must be linked as part of a larger application.
- ▶ The cFS and ROS2 packages can be dropped in place as part of a larger system. Data necessary must be made available via the software bus.

Ogma: interface

```
$ ogma fret-component-spec --fret-file-name aircraftReqSpec.json
import Copilot.Compile.C99
...

flight :: Stream Bool
flight = extern "flight" Nothing

propAvoidStall :: Stream Bool
propAvoidStall = (PTLTL.alwaysBeen (((not (flight)) && ... )))

...
```

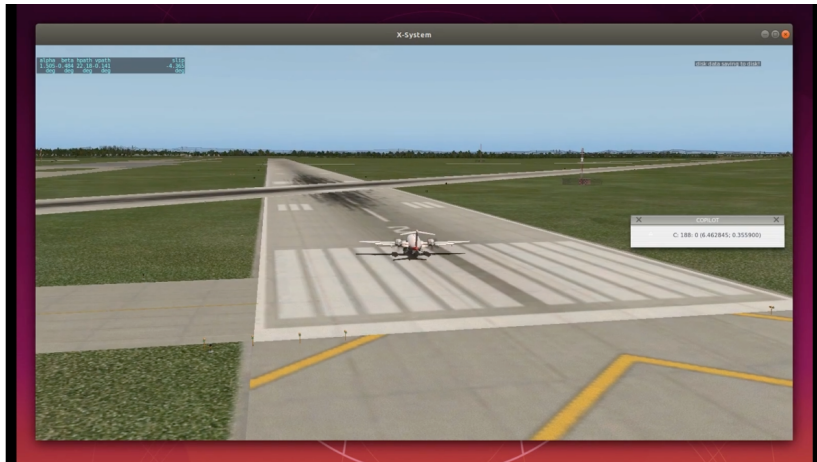
Experiments: Flights with Unmanned Vehicles



Credits: K. Darafsheh / NASA

Ogma: experiments (1)

- ▶ We used FRET, Ogma and Copilot to encode and monitor a flight simulation in X-Plane.



Ogma: experiments (1, continuation)

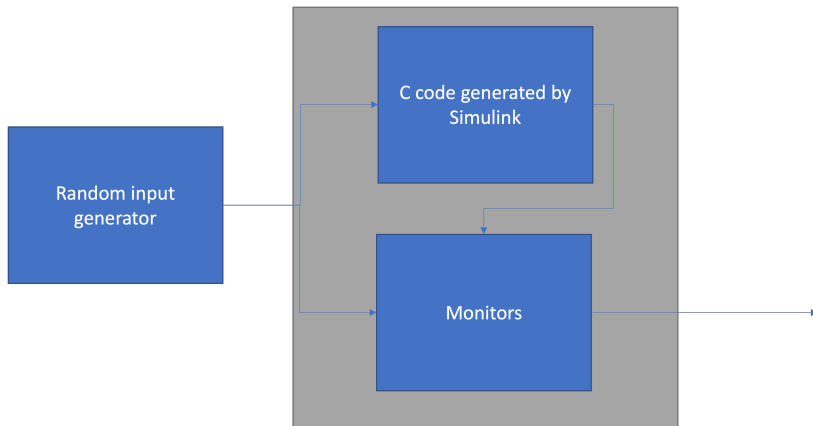
- ▶ The monitor is shown on the screen and reports any violations.



Ogma: experiments (2)

- ▶ FRET-Ogma-Copilot were used to monitor the C code generated for Simulink models implementing two of the LMCPs challenge problems: Finite State Machine, and Control Loop Regulators (REG) LMCPs.
- ▶ The random input testing system QuickCheck was used to generate random inputs and determine if any monitors reported requirement violations.
- ▶ Violations reported consistently with prior experiments using model checkers.
- ▶ Results obtained within seconds in cases where model checkers timed out.

Ogma: experiments (2, continuation)



Future

- ▶ Upcoming release: Test generation (randomized testing)
- ▶ Ongoing: Fault injection
- ▶ Ongoing: FPrime
- ▶ Ongoing: FPGAs
- ▶ Ongoing: MC/DC

References (1)

- ▶ Perez, Mavridou, Pressburger, Will, Martin, "Monitoring ROS2: from Requirements to Autonomous Robots". FMAS 2022.
- ▶ Perez, Mavridou, Pressburger, Goodloe, Giannakopoulou, "Automated Translation of Natural Language Requirements to Runtime Monitors". TACAS 2022.

References (2)

Perez, Dedden and Goodloe, "Copilot 3", NASA TM-2020-220587, 2020.

NASA/TM-2020-220587



Copilot 3

Ivan Perez

National Institute of Aerospace, Hampton, Virginia

Frank Dedden

Royal Netherlands Aerospace Center, Amsterdam, The Netherlands

Alwyn Goodloe

NASA Langley Research Center, Hampton, Virginia

References (3)

Perez, Dedden, Darafsheh, Goodloe, Pike, "A Gallery of Copilot Specifications", 2020.

Contents

1	Introduction	3
2	Simple Engine Temperature Monitor	3
3	Home Heating System	4
4	Aircraft Health Monitoring	5
5	Fault Tolerant Monitors	8
6	Well Clear	10
7	Temporal Logic	15
8	Aircraft Collision Avoidance	17

Summary

- ▶ Copilot is a high-level Runtime Verification framework that produces hard real-time C99.
- ▶ Strongly typed and uses dependent types.
- ▶ Simple connection to systems written in C.
- ▶ Ogma can help write monitors from requirements, as well as monitoring NASA cFS and ROS2 applications.
- ▶ NASA Class D (NPR7150.2).
- ▶ Used in experimental research with NASA, Galois, and others.
- ▶ Copilot, Ogma and FRET are all open source.

Acknowledgments

- ▶ Geoffrey Biggs, Guillaume Brat, Macallan Cruff, Kaveh Darafsheh, Frank Dedden, Dimitra Giannakopoulou, Alwyn Goodloe, Chris Hathhorn, Michael Jeronimo, Georges-Axel Jolayan, Jonathan Laurent, Anastasia Mavridou, Eli Mendelson, Robin Morisset, Sebastian Niller, Amalaye Oyake, Lauren Pick, Lee Pike, Will Pogge, Tom Pressburger, Patrick Quach, Ryan Scott, Kyle Smalling, Ryan Spring, Laura Titolo, Sixto Vazquez, Nis Wegmann.

Source code

- ▶ Copilot: <https://copilot-language.github.io>
- ▶ Ogma: <https://github.com/nasa/ogma>
- ▶ FRET: <https://github.com/nasa-sw-vnv/fret>

Thank you!

Runtime Verification with Oigma

Ivan Perez, Ph.D

`ivan.perezdominguez@nasa.gov`

KBR @ NASA Ames Research Center

Invited talk

University of California Santa Cruz